Supporting Software

Compilers and Autotuners
Virtual Machines
Middleware
Operating Systems

Challenges and Directions

• System software
  – Compilers
  – Middleware, libraries
  – Operating systems

• Main directions
  – Compilers vs autotuners
  – Monolithic OS vs virtual machines + libraries
Compiler Challenges

• Hard to add new optimizations to handle data/task parallelism to traditional compilers
  – Functional correctness of the code is the main concern
  – Difficult to verify optimizations against all possible cases
• For instance, Streamit is a new compiler and has been verified only on a limited set of test cases
• Autotuners?

Autotuners

• *Search-based* optimization
  – Optimize a set of library kernels by generating many variants and benchmarking each variant by running on the target platform
  – Search process tries many or all optimizations (takes hours but run only once)
• Applying autotuners to parallelism?
  – Search space is too large
  – Possible solution: decouple the search for good data layout and communication patterns from good kernels
Parallelizing compilers

• Examples
  – Streamit (streaming applications)
  – OpeMP (general purpose)
• StreamIt on the IBM cell
  – Push/pop and work are implemented using hw FIFOs managed by DMA
  – A runtime environment implement the work construct through creation and destruction of threads on the SPE engines
  – Load and remove code/data on the private memories of the SPEs

Streamit Compiler

• Main idea: parallelizing compilers such as OpenMP or HPF expose the programmer to details of parallelism, which lead to the following problems
• Granularity decisions:
  – if too small, lots of synchronization and thread creation
  – if too large, bad locality
• Load balancing decisions
  – Create balanced parallel sections (not data-parallel)
• Locality decisions
  – Sharing and communication structure
• Synchronization decisions
  – barriers, atomicity, critical sections, order, flushing
• For mass adoption, we need a better paradigm:
  – Where the parallelism is natural
  – Exposes the necessary information to the compiler
Unburden the Programmer

• Move these decisions to compiler
  – Granularity
  – Load Balancing
  – Locality
  – Synchronization
• Exploit StreamIt language features
• NB: optimization based on static TG analysis
  – No profiling
  – No real-time requirements

Properties of Stream Programs

• Regular and repeating computation
• Synchronous Data Flow
• Independent actors with explicit communication
• Data items have short lifetimes
• Benefits:
  – Naturally parallel
  – Expose dependencies to compiler
  – Enable powerful transformations
Steady State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and after executing the schedule
  - There exist a unique minimum steady state schedule
- Schedule = \{ \}

![Diagram](image1)

Steady State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and after executing the schedule
  - There exist a unique minimum steady state schedule
- Schedule = \{A\}

![Diagram](image2)
Steady State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule
- Schedule = \{A,A\}

\[\text{Schedule} = \{A,A\}\]

---

Steady State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule
- Schedule = \{A,A,B\}

\[\text{Schedule} = \{A,A,B\}\]
Steady State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule
- Schedule = \{ A, A, B, A \}

---

Steady State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule
- Schedule = \{ A, A, B, A, B \}
Steady State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and after executing the schedule
  - There exist a unique minimum steady state schedule
- Schedule = \{A,A,B,A,B,C\}

Types of Parallelism in Streamit

- Task Parallelism
  - Parallelism explicit in algorithm
  - Between filters without producer/consumer relationship
Types of Parallelism in Streamit

- **Task Parallelism**
  - Parallelism explicit in algorithm
  - Between filters without producer/consumer relationship

- **Data Parallelism**
  - Between iterations of a stateless filter
  - Place within scatter/gather pair (fission)
  - Can’t parallelize filters with state

- **Pipeline Parallelism**
  - Between producers and consumers
  - *Stateful filters can be parallelized*

Baseline 1: Task Parallelism

- Inherent task parallelism between two processing pipelines
- **Task Parallel Model:**
  - Only parallelize explicit task parallelism
  - Fork/join parallelism
- Execute this on a 2 core machine
  ~2x speedup over single core
- What about 4, 16, 1024, ... cores?
Evaluation: Task Parallelism

Baseline 1: Fine-Grained Data Parallelism

- Each of the filters in the example are stateless
- Fine-grained Data Parallel Model:
  - Fiss each stateless filter \( N \) ways (\( N \) is number of cores)
  - Remove scatter/gather if possible
- We can introduce data parallelism
  - Example: 4 cores
- Each fission group occupies entire machine
Evaluation: Fine Grained Data Parallelism

Baseline 3: Hardware Pipeline Parallelism

- The BandPass and BandStop filters contain all the work
- Hardware Pipelining
  - Use a greedy algorithm to fuse adjacent filters
  - Want # filters <= # cores
- Example: 8 Cores
Baseline 3: Hardware Pipeline Parallelism

- Resultant stream graph is mapped to hardware
  - One filter per core
- What about 4, 16, 1024, cores?
  - Performance dependent on fusing to a load-balanced stream graph

Evaluation: Hardware Pipeline Parallelism

Parallelism: Not matched to target!
Synchronization: Not matched to target!
Streamit Compiler

1. Coarsen: Fuse stateless sections of the graph
2. Data Parallelize: parallelize stateless filters
3. Software Pipeline: parallelize stateful filters

- Compile to a 16 core architecture
  - 11.2x mean throughput speedup over single core

Notes

- When # tasks > # proc a schedule must be defined within each core
- StreamIt on IBM Cell architecture
  - A runtime environment handles thread creation and destruction as well as communication through DMA
  - It implements pop/push and work directives
  - Work is the part done by the SPE engines
Phase 1: Coarsen the Stream Graph

- Before data-parallelism is exploited
- *Fuse stateless pipelines as much as possible* without introducing state
  - Don’t fuse stateless with stateful
  - Don’t fuse a peeking filter with anything upstream

**Phase 1: Coarsen the Stream Graph**

- Before data-parallelism is exploited
- *Fuse stateless pipelines as much as possible* without introducing state
  - Don’t fuse stateless with stateful
  - Don’t fuse a peeking filter with anything upstream

**Benefits:**
- Reduces global communication and synchronization
- Exposes inter-node optimization opportunities
Phase 2: Data Parallelize

Data Parallelize for 4 cores

Task parallelism!
Each fused filter does equal work
Fiss each filter 2 times to occupy entire chip

Fiss 4 ways, to occupy entire chip
Phase 2: Data Parallelize

Data parallelize for 4 cores
Task-conscious data parallelization
• Preserve task parallelism
Benefits:
• Reduces global communication and synchronization

Evaluation: Coarse-Grained Data Parallelism

Good Parallelism! 👍
Low Synchronization! 😞
Data + Task Parallel Execution

We Can Do Better!
Phase 3: Coarse-Grained Software Pipelining

- New steady-state is free of dependencies
- Schedule new steady-state using a greedy partitioning

Greedy Partitioning

To Schedule:

Cores

Time

16
Summary

- Streaming model naturally exposes task, data, and pipeline parallelism
- This parallelism must be exploited at the correct granularity and combined correctly